OMiSCID 2.0, un intergiciel gratuit pour la construction d'applications distribuées

Rémi Barraquand, Dominique Vaufreydaz, Rémi Emonet, Amaury Nègre, Jean-Pascal Mercier, Patrick Reignier

MajecSTIC 2010
Bordeaux, France, du 13 au 15 octobre 2010

OMiSCID 2.0, un intergiciel libre et opensource pour la construction d'applications ubiquitaires

Rémi Barraquand, Dominique Vaufreydaz, Rémi Emonet, Amaury Nègre, Jean-Pascal Mercier et Patrick Reignier

Équipe PRIMA
Laboratoire d'Informatique de Grenoble (CNRS, UPMF, UJF, INRIA)
INRIA Grenoble Rhône-Alpes
655, avenue de l'Europe 38334 Saint Ismier Cedex
Contact : omsicid-info@inrialpes.fr

Résumé

Cet article présente OMiSCID et ses dernières évolutions vers la version 2.0. OMiSCID est un intergiciel facilitant le développement et le déploiement d'application réparties et notamment des applications ubiquitaires dans les environnements intelligents. OMiSCID est entièrement gratuit, libre et opensource avec une licence non collante de type MIT [1]. Il est utilisable avec plusieurs langages de programmation et sous différents systèmes d'exploitation. Son interface de programmation orientée utilisateur tend à fournir une simplicité d'utilisation maximale et une courbe d'apprentissage minimale. L'objectif de cet article est d'exposer les différentes fonctionnalités offertes par OMiSCID en illustrant son utilisation au travers d'exemples concrets. Nous présentons également OMiSCID GUI, une plateforme générique offrant une interface graphique facilitant le développement, le débuggage et la construction d'application.

Mots-clés : intergiciel, applications distribuées, architecture à service, découverte de service, informatique ambiante

1. Introduction

Le rôle de l'informatique dans nos sociétés n'a fait qu'évoluer au cours des dernières décénies. Les avancées technologiques ont transformées le paysage informatique avec l'apparition de petits dispositifs à bas prix tels que les assistants personnels, les téléphones mobiles, les webcams, etc. L'utilisation classique de l'ordinateur personnel (application centralisée, clavier, souris) a alors évoluée vers une informatique distribuée ou ambiante. L'objectif de l'informatique ambiante [11] est de faire disparaitre l'informatique traditionnelle au profit d'un espace informatisé, au service de l'usagé, dans lequel interagissent une multitude de dispositifs hétérogènes en constante évolution et apparaissant dynamiquement : un environnement intelligent. Cet espace informatisé doit prendre en compte la mobilité, la multiplicité des plateformes et percevoir le contexte afin de mieux comprendre et anticiper les besoins de l'utilisateur et être en mesure de proposer automatiquement des services appropriés.

Ces évolutions ont nécessité la mise en œuvre d'architectures logicielles non plus monolithiques mais distribuées sous forme de composants répartis sur différentes plateforme et pouvant communiquer entre eux. Toutes les briques logicielles doivent pouvoir collaborer pour faire émerger une application distribuée. C'est le rôle de ce que l'on nomme un intergiciel (*middleware*), une couche logicielle intermédiaire permettant de simplifier la tâche des développeurs. Au cours des dernières années, plusieurs projets de recherche ont été initiés dans ce cadre afin de gérer les équipements domestiques et multimédias [5], l'énergie d'un bâtiment [12] ou d'aider au maintien des personnes âgées à domicile [1, 10].

Nous présentons dans cet article, O^{3}MiSCID (plus simplement nommé OMiSCID), un intergiciel opensource, libre et gratuit proposant une programmation orientée service (*Service Oriented*

1. Il est disponible et téléchargeable depuis son site http://omiscid.gforge.inria.fr/

Architecture - SOA). OMiSCID signifie Object Oriented Opensource Middleware for Service Communication Inspection and Discovery.

Dans un premier temps, nous présenterons le contexte qui a donné naissance à OMiSCID . Nous aborderons ensuite les bases de celui-ci et son implémentation 2.0. Avant de conclure, nous présenterons son interface graphique et les possibilités qu'elle offre tant au niveau développement, débuggage que dans la construction d'applications.

2. Besoin des applications distribuées ou ubiquitaires

La construction d'applications réparties, plus particulièrement au sein d'équipes de recherche ou de projets internationaux, nécéssite de lever des verous concernant la coopération et/ou la composition de composants logiciels fortement hétérogènes. Les besoins particuliers de chacune des équipes font que celles-ci ont généralement peu ou pas de points de concordance, que se soit en terme de language de programmation ou d'environement de travail. Ainsi, certaines utilisent Java^{tm} pour l'acces à des libraries spécifiques ou par soucis de portabilité, d'autres C++ pour des questions de performance, Windows^{tm} pour le support d'une carte particulière ou encore Linux/-MacOs pour d'autre critères. Dans ce contexte, une architecture à services facilite l'interconnection de composants. Cependant, le fait de combiner des composants hétérogènes reste une problèmatique ouverte. Pour mettre en oeuvre ce type d'architecture, les problématiques à adresser sont de natures complexe et diverse :

- *muti-plateforme* et *muti-langage*. L'exécution doit pouvoir se faire sur plusieurs systèmes d'exploitation et dans différents langages de programmation ;
- *recherche de services*. Il est nécessaire d'avoir en permanence une liste des services qui s'exécutent sur les différentes machines présentes dans l'environnement et de gérer leur disparition et leur apparition : c'est ce que l'on appelle la découverte de service.
- *transport d'informations*. L'intergiciel doit avoir la capacité de gérer à la fois les aspects pair-à-pair et les flux de traitement des données. Ces données allant de messages courts à des flux en provenance d'une caméra par exemple (voir figure 1).
- *mise au point et surveillance*. Il est important dans un tel intergiciel de disposer d'outils de surveillance et de mise au point.

Si l'on s'intéresse aux solutions disponibles, nous pouvons dégager trois grandes tendances. La première sont les Web Services [9]. Ceux-ci utilisent des technologies du Web (le protocol HTTP par exemple) pour mettre en place des architectures à services distribuées. Ils ont le gros avantage d'être utilisables dans de nombreux langages. En utilisant *Web Services Dynamic Discovery* (*WS-Discovery*) et *Web Services Description Language* (WSDL), le raisonnement et la découverte de services sont possibles. Cependant, ceux-ci sont peu adaptés aux transports de gros flux de données et sont, par nature, difficiles à débogguer. Nous n'avons donc pas retenu cette solution.

La seconde tendance est la branche OSGi^{tm} [2] qui regroupe une large communauté très active. Il permet l'instanciation local sur une machine et l'interconnexion de composants. Il est possible d'utiliser des outils comme iPOJO [6] et R-OSGi [3] pour simplifier la tâche du développeur dans son utilisation. Le premier permet de s'affranchir de l'écriture de code fastidieux pour la gestion des dépendances ou encore la configuration du composant. Le second fournit une surcouche à OSGi lui permettant de fonctionner de manière distribuée. Malgré cela, le principal reproche que nous pouvons faire aux approches OSGi est leur fort couplage au langage Java ce qui fait que nous ne les avons pas retenues [3].

La dernière tendance concerne les intergiciels dédiés à une tâche spécifique et, bien souvent, à un type d'environnement donné. Ils sont très nombreux mais nous pouvons citer en exemple smart-flow [7] qui a été très largement employé dans des projets européens et internationaux en traitement du signal acoustique. Sa force est également sa faiblesse : sa performance à gérer des graphes de flux mutimédias à fort débit n'a d'égale que son inaptitude à transférer d'autres types de données et sa configuration complexe.

Le constat d'absence d'une solution couvrant tous les besoins que nous avons exposés a été le point de départ de la naissance d'OMiSCID .

2. Voir http://www.osgi.org/.
3. OMiSCID propose une version OSGi de son pendant Java.

3. Concepts OMiSCID

OMiSCID est basé sur trois concepts principaux qui permettent de découvrir et interconnecter dynamiquement des composants logiciels présents sur le réseau : le service, le connecteur et la variable. Un service est un composant logiciel proposant justement un service sur le réseau. Un connecteur est un port réseau pour échanger des informations avec un service. Une variable permet d'exposer des valeurs contenues dans un service. Nous allons maintenant détailler ces concepts.

3.1. Services

Comme nous l'avons déjà mentionné, la philosophie d'OMiSCID prend son inspiration dans les architectures dites à services (SOA en anglais). Un service est un composant logiciel qui expose, de manière transparente et légère, les fonctionnalités qu'il met à disposition pour assurer une tâche particulière. Les fonctionnalités exposées par un service sont donc visibles et accessibles par tout autre service sans se soucier de contraintes d'implémentation. Dans notre cas, un service expose ses fonctionnalités par l'intermédiaire de variables et de connecteurs. Il est composé par défaut d'un ensemble de variables de description :

- Un nom. Cette variable doit correspondre à la fonction remplie par le service et être, de préférence, lisible par un humain, par exemple *Caméra* ;
- Une classe. Cette variable permet de regrouper les services dans des catégories, par exemple *VideoProcessing* ;
- Un identifiant unique. Cet identifiant permettant de distinguer les services entre eux est généré automatiquement par OMiSCID .
- Le nom de la machine. C'est le nom de l'ordinateur où s'exécute le service ;
- Un propriétaire. Cette variable correspond au non de l'utilisateur qui a déployé le service.

Pour un service donné, OMiSCID fournit également la liste et la description de toutes les variables et de tous les connecteurs qui lui sont propres. Ces informations permettent aux autres services de le découvrir (voir section 4.2.2) et d'échanger des données avec lui.

La granularité des services est totalement dépendante de la volonté des développeurs. Cependant, nous conseillons que celle-ci soit la plus fine possible pour permettre une plus grande réutilisabilité et une maintenance plus aisée. Lors de nos travaux de recherche, nous avons mis en place de nombreux services répondant à ces considérations et offrant des fonctionnalités soit aux autres services, soit aux utilisateurs. Parmi ces services nous pouvons citer en exemple un service de capture vidéo qui diffuse en temps réel le flux vidéo issu d'une caméra, un service réalisant des traitement d'images ou plus simplement un service qui joue en synthèse vocale du texte reçu par le biais d'un connecteur.

3.2. Connecteurs

Les connecteurs sont des ports de communication qu'il est possible de créer pour permettre les échanges de données entre les services. Ces communications se font par le biais d'envoi de messages de taille quelconque contenant des informations binaires ou sous forme texte. Chaque envoi peut être destiné à un service en particulier où à tous les services connectés en même temps. OMiSCID garantit que l'ordre d'arrivée des messages respecte l'ordre d'émission. La communication peut être effectué avec les protocoles TCP ou UDP. Pour ce dernier, en cas de perte d'un message, une notification est adressé aux deux pairs.

Un service peut disposer de plusieurs connecteurs permettant une séparation logique des données en fonction de leur provenance. Chaque connecteur est indépendant et défini par un nom, une description et un port permettant de s'y connecter. Il est également associé à un type : entrée, sortie ou entrée/sortie. Ainsi, un connecteur de type entrée/sortie sera plutôt utilisé dans les communications pair à pair. Les connecteurs entrée et les connecteurs sortie pourront être employés dans la définition d'un flux de traitement des données. Pour illustrer ce principe, nous pouvons nous intéresser au cas de la mise en place distribuée d'une partie du caméraman automatique de l'équipe PRIMA [8]. Ce système utilise plusieurs caméras et microphones pour analyser ce qui se déroule dans un environnement et réaliser un montage automatique de la meilleure séquence vidéo possible (meilleur angle de vue, meilleure source sonore, choix des transitions, ...).

Pour cela, il coordonne, en fonction de ce qui est disponible dans l'environnement, différents ou-

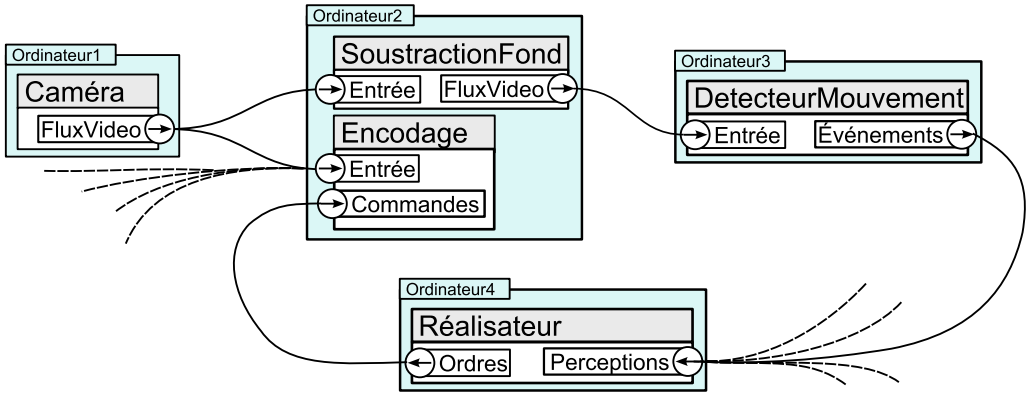

FIGURE 1 – Exemple de flux de données dans une application OMiSCID

tils de perception allant de la différence de fond à un système de localisation en trois dimensions de personnes en passant par un système de détection de la parole ou de localisation acoustique. Le service réalisateur agit comme contrôleur sur le service *Encodage* afin de choisir d'enregistrer les flux provenant de la meilleure caméra et des meilleurs microphones.

3.3. Variables

Les variables sont des éléments de description des services et de leur état. Il est possible de définir un nombre quelconque de variables. Celle-ci sont définies à l'aide de différents attributs :

- Un nom. Le nom est une chaine de caractère d'une longueur maximal de 254 caractères.
- Une description. La description est une chaine décrivant brièvement la variable.
- Un type. Le type est indiqué sous forme de texte. Il permet de pouvoir analyser le contenu de la variable.

Les services distants peuvent demander à être notifié de tout changement de valeur d'une ou plusieurs variables.

4. Notions Avancées

4.1. Multiplateforme/Cross-Language

Conformément aux requis que nous avons présentés dans la section 2, OMiSCID supporte plusieurs langages de programmation : Java, C++ et le langage de script Python. L'implémentation Java est de plus utilisable depuis Matlab et depuis les nombreux languages tournant sur la machine virtuelle Java (JavaFX, scala, groovy, javascript, etc.). Cela permet de couvrir un assez vaste champ de possibilités tout en laissant la possibilité de faire des "wrappers" pour d'autres langages. C'était d'ailleurs le cas pour la partie Python jusqu'à l'arrivée de la version pure Python (pyOMiSCID). L'unification de l'API a été faite dans le respect des facilités de programmation de chacun des langages pour ne pas changer les habitudes des développeurs ou contraindre leur utilisation. Concernant les systèmes d'exploitation, OMiSCID supporte les versions 32 bits et 64 bits de Windows (2000 et suivant), de Linux et de Mac OSX.

4.2. Interface de programmation unifiée (*User Friendly API*)

Afin d'optimiser les échanges entres les utilisateurs d'OMiSCID , nous avons décidé d'unifier l'interface de programmation (API) dans les différents langages supportés. Cela a été fait dans le respect des facilités de chacun de ces langages dans le but de ne pas resteindre ou changer les habitudes des développeurs. Cette API unifiée apporte un avantage : les utilisateurs n'ont qu'une seule API à apprendre et peuvent échanger entre eux même s'ils n'emploient pas le même langage de programmation. Nous l'illusterons grâce à des extraits de code en C++ correspondant au service *Caméra* de la figure 1. Pour simplifier notre discours et nos exemples, nous considérerons que celui-ci est à l'origine des recherches de services et des connexions avec ses pairs.

4.2.1. Création de service

La base du fonctionnement d'OMiSCID est le service. La création d'un service consiste simplement à invoquer l'usine à services (*ServiceFactory*) et de lui demander de créer un service :

```
1 #include <ServiceControl/UserFriendlyAPI.h>
2 using namespace Omiscid;
3 Service * pServ = ServiceFactory.Create( "Caméra" );
4 pServ->AddConnector( "Flux Vidéo", "Flux vidéo de la caméra", AnOutput );
5 pServ->Start();
```

Après les prémices de rigueur, à la ligne 3, nous demandons à la *ServiceFactory* de créer un service *Caméra*. Nous ajoutons à ce service un connecteur nommé *FluxVideo* avec un commentaire indiquant sa fonction. Le dernier paramètre indique que ce connecteur est en sortie uniquement. Enfin, nous démarrons le service ce qui a pour effet de l'enregistrer sur le réseau et de le rendre disponible aux autres services.

4.2.2. Filtres et découverte de service

Lorsque qu'un service requière d'autres services pour fonctionner, il lui est nécessaire de pouvoir rechercher l'ensemble des services compatibles avec ses besoins. Pour faire cela, OMiSCID permet de lister tous les services disponibles et de filtrer ces services à l'aide de filtres de recherche. Un filtre est un ensemble de conditions (nom du service, présence d'une variable ou d'un connecteur, valeur d'une variable, etc.) reliés par des opérateurs booléens. Si ceux-ci ne sont pas suffisants, il est possible de créer des filtres personnalisés.

En utilisant ces filtres, il est possible d'obtenir la liste de tous les services remplissant les conditions spécifiées à un instant donné. Il est possible aussi de s'enregistrer pour recevoir des notifications à chaque fois qu'un nouveau service correspondant au filtre apparaît ou disparaît.

```
1 MyServiceListener SearchNeededServices;
2 ServiceRepository * pRep = ServiceFactory.CreateServiceRepository();
3 pRep->AddListener( SearchNeededServices,
4       Or(And(NameIs("SoustractionFond"),HasConnector("Entrée",AnInput)),
5          And(NameIs("Encodage"),Not(HostIs(GetLocalHostName())))  );
```

Dans cet exemple, nous créons un objet *SearchNeededServices* dont la classe *MyServiceListener* est obtenue par héritage de la classe *ServiceRepositoryListener* (non détaillé ici). Nous demandons ensuite à l'usine à service de nous fournir un dépôt pour contenir les informations des services qui correspondent au filtre passé en paramètre : les services *SoustractionFond* qui ont un connecteur d'entrée *Entrée* et tous les services *Encodage* qui ne s'executent pas sur la machine locale. À l'apparition ou à la dispatition de tels services, les méthodes *ServiceAdded* et *ServiceRemoved* de *SearchNeededServices* seront invoquées, nous permettant ainsi de savoir si nous avons les services nécessaires pour fonctionner.

4.2.3. Gestion des communications

Nous allons maintenant aborder les aspects connexion et gestion de l'envoi et de la réception de message. Nous venons de voir comment obtenir la liste des services qui nous intéressent. Le dépôt de la section précédente nous permet d'obtenir des objets *ServiceProxy* : *ProxySF* pour la soustraction de fond et *ProxyEnc* pour l'encodeur.

```
1 pServ->ConnectTo( "Flux Vidéo", ProxySF, "Entrée" );
2 pServ->ConnectTo( "Flux Vidéo", ProxyEnc, "Entrée" );
3 // ...
4 pServ->SendToAllClients( "Flux Vidéo", Image, TailleImage );
```

Sur les lignes 1 et 2, nous utilisons les ServiceProxy pour connecter notre connecteur *FluxVidéo* sur les connecteurs *Entrée* de chacun des services. Ensuite, à chaque nouvelle image disponible, il suffit de demander à OMiSCID d'envoyer celle-ci à tous les services connectés (ligne 4).

Concernant la réception de messages, la mécanique est la même que pour la notification des services. On crée un objet *callback* en héritant d'une classe spécifique d'OMiSCID et on enregistre cet objet pour recevoir les messages en provenance d'un ou plusieurs connecteurs.

4.3. Messages structurés et appels de procédures distantes

Les messages structurés permettent de faciliter les échanges de messages en garantissant le support des types de bases (entier, flottant, chaîne de caractère, ...), le support des tableaux ainsi que

la sérialisation d'objets complexes en cross-langage. Pour permettre cela, nous avons tout d'abord choisi d'utiliser le format JSON [4] pour encoder les données. Le JSON est un langage textuel qui permet de représenter de l'information structurée et qui est très facile à interpréter par un humain et par une machine (faible coût de calcul). Ces types de données sont suffisamment génériques et abstraits pour, d'une part, pouvoir être représentés dans n'importe quel langage de programmation, d'autre part, pouvoir représenter n'importe quelle donnée concrète. Nous avons ajouté des mécanismes de sérialisation simples pour n'importe quel type d'objet et de variable. Dans le cas des gros flux de données binaires, il est préférable de transmettre les données brutes pour des raisons de performance.

OMiSCID 2.0 offre également la possibilité à un service d'exposer des fonctionnalités sous forme de méthodes appelables à distance (mécanisme de *Remote Procedure Call* - RPC). Pour permettre l'appel de procédures à distance, nous nous appuyons sur les messages structurés pour échanger les paramètres et le résultat. Les mécanismes d'enregistrement et d'appel de méthodes sont transparents pour l'utilisateur : il suffit d'associer des méthodes à un connecteur, les rendant ainsi invocable à distance. Lors d'un appel de méthode à distance, un objet *handler* est retourné instantanément. Celui-ci permet de récupérer le(s) résultat(s) de l'appel de deux manières différentes. Une utilisation bloquante permettant une attente passive du résultat et une méthode non bloquante permettant de récupérer le résultat par l'intermédiaire d'une *callback*. Ce mécanisme offre une liberté d'implémentation au développer lui permettant d'effectuer des appels synchrones et asynchrones, fonctionnalité n'étant pas offerte dans les paradigmes de RPC classiques.

5. Interface Utilisateur

OMiSCID propose une solution simple pour déclarer et rechercher des services. Pour un développeur, lors de la mise en place des differents services dans un environment complexe où fourmille déja un écosystème varié, il est de mise de pouvoir visualiser l'environement, surveiller les interactions ainsi que d'en contrôler les différents composants. À ces tâches s'ajoute bien sûr le deboggage de services qui peut rapidement, sans les outils appropriés, devenir un exercice difficile. Pour faire face aux exigences des utilisateurs, nous avons développé OMiSCID Gui, que nous présentons brièvement ici.

OMiSCID Gui est basé sur la plateforme Netbeans écrite en Java. Il hérite donc des fonctionalités de cette plateforme : portabilité, modularité, gestion avancée des fenêtres, etc. L'aspect modulaire a été exploité pour créer une application que les développeurs peuvent facilement enrichir. Ainsi OMiSCID Gui est composé d'un cœur, très petit, et d'un grand nombre de modules complémentaires (ou plugins) développés par diverses personnes.

Le cœur d'OMiSCID Gui propose, dans ce menu, des actions pour manipuler les connecteurs (visualiser les messages émis et/ou envoyer des messages) et les variables (surveiller les changements de valeurs et/ou émettre des requêtes de changement). Ces fonctionalités de base fonctionnent avec tout service OMiSCID existant mais sont génériques (e.g. le message brut est affiché). Le menu contextuel de l'arbre de services est dépendant de la sélection et peut être enrichi par les plugins. Ce menu est le point d'extension principal d'OMiSCID Gui. Pour illustrer les possibilités offertes par OMiSCID Gui, nous présentons certains plugins existants et disponibles au travers du site de mise à jour automatique.

- Un plugin sert à tracer dynamiquement des courbes de l'évolution temporelle de variables.
- Un plugin permet de contrôler de manière synchrône plusieurs services vidéos rejouant chacun une vidéo pré-enregistrée. Ces outils de visualisation sont précieux pour le deboggage et sont aussi utilisés à des fins de démonstration.
- Un plugin permet de visualiser tous les services et leurs interconnections sous forme de graphe.
- De nombreux autres plugins sont également disponibles : écoute ou visualisation de flux audio, pilotage de caméras ou vidéoprojecteurs pilotables, visualisation 3D pour les système de suivi vidéo 3D, etc.

OMiSCID Gui est riche en fonctionalités pour les utilisateurs d'OMiSCID. De plus, sa conception modulaire mettent à la portée de tout développeur la création de nouveaux plugins. Les facilités de déploiement et de mise à jour proposées par OMiSCID Gui en font une plateforme idéale pour

4. Voir http://www.json.org/.

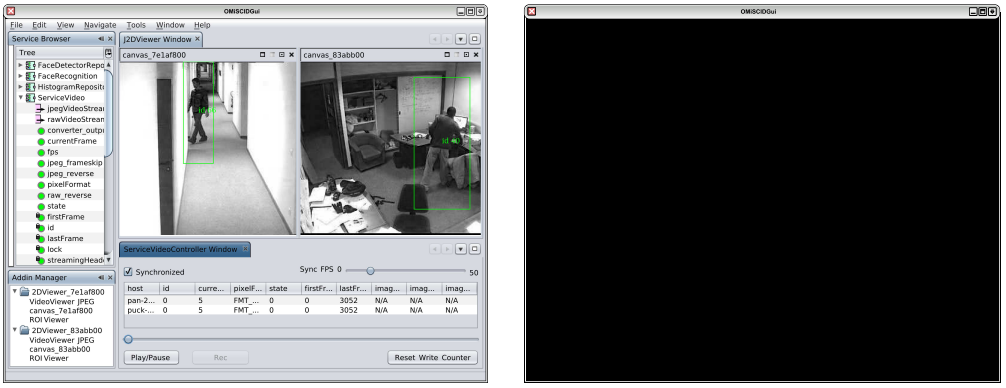

FIGURE 2 – Exemples de sessions OMiSCID Gui lors de la mise au point d'un système de suivi multicaméra.

le développement de démonstations et d'applications utilisant des services.

6. Performances

Nous présentons dans cette section des tests d'OMiSCID faites sur la grille de calcul Grid5000 [2]. L'environnement utilisé consiste en un ensemble de 10 machines octo-cores interconnectées par un réseau 1 gigabits du cluster genepi de Grenoble. Le système d'exploitation utilisé est une distribution Linux Debian en 64 bits, le langage de programmation le c++. Pour d'autres évaluations d'OMiSCID, le lecteur pourra se référer à [4].

6.1. Applications

L'utilisation d'OMiSCID introduit des effets sur les applications. OMiSCID utilise plusieurs processus légers (*thread*). Le nombre de ces processus légers est corrélé au nombre de services et à la complexité, en terme de connecteurs, de chaque application. Concernant l'utilisation mémoire, la surcharge de mémoire introduite est faible. Ainsi, le chargement des librairies et la gestion d'un premier service avec 2 connecteurs et 10 variables rajoute 1,2 Mo de mémoire. Il y a aussi 6 processus légers de créés pour assurer la gestion asynchrone des communications. L'ajout d'un service du même type accroit à chaque fois la mémoire d'environ 800 Ko et de 6 processus légers.

6.2. Débit réseau

OMiSCID permet la transmission des données entre des services. L'un des points intéressants est de connaître quel est le débit utilisateur, sachant qu'OMiSCID, pour ses besoins internes de fonctionnement, rajoute des informations aux données transmises. Pour ce test, notre configuration n'utilise que 2 machines de notre set. Le débit utilisateur est calculé après l'envoi de 1 Go de données entre 2 services lancés, chacun sur l'une des machines. Cette expérience montre qu'il est possible de transmettre jusqu'à 112,48 Mo/s (899 Mbits/s). En comparaison, l'utilitaire linux *nttcp* donne des performances entre les 2 mêmes machines de 118.87 Mo/s (950 Mbits/s). Le protocole de communication d'OMiSCID ainsi que la gestion des communications sont suffisamment performants pour permettre la transmission de tous types de données.

6.3. Découverte de services

La découverte de service est l'une des fonctionnalités clefs d'OMiSCID. C'est également une fonctionnalité coûteuse en temps puisque la découverte se fait au traver du reseau sur des services distribués. Pour notre test, nous nous sommes placés dans la situation la plus contraignante. Nous avons lancé 50 services sur chacune des 9 premières machines de notre environnement de test, soit 450 services. La dernière machine recherche aléatoirement l'un de ces services en fonction de la valeur d'une de ses variables. Pour 10000 recherches, le temps moyen de recherche constaté est de 223 ms. L'écart-type est de 119, le minimum de 2 ms et le maximum de 455 ms.

Le temps de découverte n'est cependant pas linéaire en fonction du nombre de services recherchés. L'algorithme de recherche est optimisé et ne parcours qu'une fois la liste des services présents. Ainsi, la recherche de 25 services parmi les 450 sur le réseau prend en moyenne 480 ms.

7. Conclusion

Nous avons présenté OMiSCID 2.0, un intergiciel facilitant la construction d'applications ubiquitaires et réparties, offrant les avantages suivant :

- une architecture claire et flexible manipulant des concepts simples et intuitifs ;
- une API simple et unifiée disponible dans plusieurs langage de programmation et multiplateforme ;
- une interface modulaire, extensible basé sur la plateforme Netbeans ;
- une solution performante offrant une découverte de service avancé et paramétrable ainsi qu'un passage à l'échelle avéré ;
- un intergiciel libre et opensource.

OMiSCID a joué un rôle central dans différents projets nationnaux et internnationnaux. Parmi ces projets, nous pouvons citer le projet CAPSER [1] visant à la mise au point d'un système complet de maintien de personnes âgées à domicile. OMiSCID a rempli son rôle fédérateur permettant une intégration rapide avec des outils de nos partenaires académiques et industriels. Nous l'avons également employé dans des expérimentations de type Magicien d'Oz pour évaluer le rôle et l'importance de la compréhension mutuelle entre les usagés et les bâtiments intelligents dans un scénario d'apprentissage. OMiSCID et OMiSCID Gui ont permis de gérer efficacement la multitude de services déployés, leurs interconnexions, leurs communications mais aussi a permis de proposer une solution performante et redéployable dans différents site d'expérimentation.

Bibliographie

1. Site du projet ANR CASPER : http://www-prima.imag.fr/casper/.
2. F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jégou, S. Lanteri, N. Melab, R. Namyst, P. Primet, O. Richard, E. Caron, J. Leduc, and G. Mornet. Grid'5000 : a large scale, reconfigurable, controlable and monitorable Grid platform. In *6th IEEE/ACM International Workshop on Grid Computing - GRID 2005*, Seattle, USA, 11 2005. Grid 2005 held in conjunction with SC'05, the International Conference for High Performance Computing, Networking and Storage.
3. W. Dejun, H. Linpeng, W. JianKun, and X. Xiaohui. Dynamic software upgrading for distributed system based on r-osgi. In *CSSE '08 : Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, pages 227–231, Washington, DC, USA, 2008. IEEE Computer Society.
4. R. Emonet. *Semantic Description of Services and Service Factories for Ambient Intelligence*. PhD thesis, Grenoble INP, sep 2009.
5. C. Escoffier, J. Bardin, J. Bourcier, and P. Lalanda. Developing User-Centric Applications with H-Omega. In *Mobile Wireless Middleware, Operating Systems, and Applications - Workshops*, pages 118–123. Springer Berlin Heidelberg, April 2009.
6. C. Escoffier, R.S. Hall, and P. Lalanda. ipojo : an extensible service-oriented component framework. *Services Computing, IEEE International Conference on*, 0 :474–481, 2007.
7. A. Fillinger, L. Diduch, I. Hamchi, M. Hoarau, S. Degre, and V. Stanford. The nist data flow system ii : A standardized interface for distributed multimedia applications. In *World of Wireless, Mobile and Multimedia Networks, 2008. WoWMoM 2008. 2008 International Symposium on a*, pages 1 –3, 23-26 2008.
8. F. Metze, P. Gieselmann, H. Holzapfel, T. Kluge, M. Wolfel, I. Rogina, A. Waibel, J. Crowley, P. Reignier, D. Vaufreydaz, F. Bérard, B. Cohen, J. Coutaz, S. Rouillard, V. Arranz, M. Bertran, and H. Rodriguez. The fame interactive space. In *2nd Joint Workshop on Multimodal Interaction and Related Machine Learning Algorithms*, page 4, Edinburgh - UK, feb 2005.
9. M. Papazoglou. *Web Services : Principles and Technology*. Prentice Hall, 2007.
10. A.L. Rodrigues, I. Gomes, L. Bezerra, A. Sztajnberg, S. Carvalho, A. Copetti, and O. Loques. Using discovery and monitoring services to support context-aware remote assisted living applications. *Computational Science and Engineering, IEEE International Conference on*, 2 :1092–1097, 2009.
11. M. Weiser. The computer for the 21st century. *Scientific American*, February 1991.
12. L.W. Yeh, Y.C. Wang, and Y.C. Tseng. ipower : an energy conservation system for intelligent buildings by wireless sensor networks. *Int. J. Sen. Netw.*, 5(1) :1–10, 2009.